Dynamic Function Specialization

Arif Ali Ap, Erven Rohou

Dynamic Function Specialization

Arif Ali AP
Inria
Campus de Beaulieu
35042 RENNES CEDEX
France
Email: arif-ali.ana-pparakkal@inria.fr

Erven Rohou
Inria
Campus de Beaulieu
35042 RENNES CEDEX
France
Email: erven.rohou@inria.fr

***Abstract*—Function specialization is a compilation technique that consists in optimizing the body of a function for specific values of an argument. Different versions of a function are created to deal with the most frequent values of the arguments, as well as the default case. Compilers can do a better optimization with the knowledge of run-time behaviour of the program. Static compilers, however, can hardly predict the exact value/behaviour of arguments, and even profiling collected during previous runs is never guaranteed to capture future behaviour. We propose a *dynamic function specialization* technique, that captures the actual values of arguments during execution of the program and, when profitable, creates specialized versions and include them at runtime. Our approach relies on dynamic binary rewriting. We present the principles and implementation details of our technique, analyze sources of overhead, and present our results.**

I. Introduction

An optimizing compiler not only transforms human readable high level program to machine readable low-level binary code but also applies optimizations to improve the program, e.g. to reduce its execution time, its energy consumption or its code size. This optimizing functionality of compilers makes programming easier because developers can focus on features and readability, and let performance for the compiler to deal with. Nowadays, state-of-the-art compilers provide a great number of optimizations, e.g., GCC 5.3.1 enables 131 passes at optimization level `-O3`.

Due to its nature, a static compiler has a limited visibility when it comes to taking into account the dynamic environment or behaviour of an application. Unknowns include actual input data that impacts the values flowing through the program, or hardware-specific details (independent software vendors building software in production cannot distribute fully optimized programs because the end-users run them on a variety of hardware platforms).

Such difficulties can be overcome by *Dynamic Optimization*, since it is carried out during the execution of the program. The different use cases of dynamic optimization are discussed in previous works [16], [15], [11], [12], [19], [20].

Function Specialization (also known as *Procedure Cloning*) is one of such optimization techniques applied to the functions in a program based on its parameters [3]. In this technique, different versions of a function are created according to the most frequent values taken by its parameters. In the case of function specialization, it is also often difficult to predict/know during the static compilation phase the argument value/behavior. To create specialized versions of a function at static compilation phase, the compiler needs to assume or predict some values or some common behavior to the parameters which might not be feasible in many cases. But that is not the case with dynamic optimization where the actual values or behavior of arguments are known. In our proposed *Dynamic Function Specialization*, we apply function specialization at run time. The specialized versions of the function are created, according to the actual value of parameters, during the execution of the process.

A more detailed explanation of function specialization and its scopes are given in Section II. Section III gives a general idea about dynamic function specialization and a detailed explanation of our implementation is provided in Section IV. Section V illustrates our approach with a simple example. Section VI shows the experimental setups and the result of applying our idea on different benchmarks. And finally, Section VII discusses related work, and Section VIII concludes our work.

II. Function Specialization

Function specialization is one of the optimization techniques used to reduce the execution time of a function. The idea is that instead of calling a generic function for all the call sites, call different versions of it according to the values taken by the parameters. The call sites of a function are divided into groups based on the values taken by the parameters, and a specialized version of the function is produced for each group [3]. Each version is specially optimized for one or particular category of arguments so that they are expected to run faster as compared to the original generic version for such arguments. For example, consider the function `Foo` in Listing 1. All call sites of `Foo`, where value of parameter `b` is a power of `2`, can be considered as one group and a specialized version `Foo_b2n` can be produced for it. Now, `Foo_b2n` can execute faster compared to `Foo` when $b = 2^n$, because of the use of `shift` operator instead of more expensive `division` operator used in `Foo`. So the call `Foo(a,8)` can then be replaced by a call to `Foo_b2n` as `Foo_b2n(a,3)`.

This work is partially funded by the French Ministry of Industry under grant Nano2017 PSAIC.

Listing 1 Function Specialization

```
Foo(a, 8);
double Foo(unsigned int a[], unsigned int b)
    for i = 1 to  100
        c += a[i]/b;
return c;
        (a) Original code
  ----------------------------------
Foo_b2n(a,3);
double Foo_b2n(unsigned int a[], int n)
    for i = 1 to  100
        c += a[i]>>n;
return c;
 (b) Specialized code when b = pow(2,n).
```

For applying function specialization, knowing the value of the parameter(s) is the key. In most of the cases, the function calls do not contain constants as arguments, instead they have variables, like in `Foo(a,x)`. In such cases, it is not straight forward to do function specialization since the values of variables might be unknown. With a profile-guided optimization, having a simulated execution of the code during compilation for knowing the behaviour of the program, the value or property of the parameters can be predicted. However it may not be a feasible solution all the time because the predicted behaviour can vary at actual run time. Therefore, applying function specialization during static compilation phase is very difficult, which essentially means specialization would be more effective when applied dynamically by knowing the exact values of variables.

Knowing the value of a variable provides scope for various optimizations, such as *constant propagation, constant folding, algebraic simplification, strength reduction, unreachable code elimination, short circuiting, loop unrolling, vectorization* etc. [14], [3]. Optimizations like *constant propagation, constant folding* and *algebraic simplification* allows an expression in the program to be precomputed, which can speed up the execution of the program. Similarly, expensive operators could be replaced by equivalent less expensive operators by applying *strength reduction*. In some cases where the trip count of loops depends on the parameter value, the trip count can be precalculated and so *Loop unrolling* and *vectorization* could be more effective.

Such repetition of values may be surprising at first glance. Analyzing the reasons behind this behavior is beyond the scope of this paper. However, we note that it has been observed before, and this is not the sign of poor software or compiler. Modular software engineering and code reuse contribute to this phenomenon, as well as underlying semantics of the data being processed (modeling the real world).

III. Dynamic Function Specialization

Dynamic function specialization is another optimization technique in which function specialization is applied on a running program to improve its execution time. In this technique the different versions of the function are created dynamically according to the live values taken by its parameters. Since knowing the actual value of arguments is very important to perform function specialization, it will be more effective if applied on a program in execution. Further, a more hardware specific optimized version can be produced in this technique due to the knowledge of running hardware platform, compared to static function specialization technique.

A. Use case

Dynamic function specialization is useful when the static compiler is unable to extract/identify the values taken by the parameters of a function to create the specialized versions statically. A specialized version of a function is needed when there is a good chance of calling the specialized version. That is, the probability of repeating at least one of its parameters should be high. Some functions do repeat their arguments, but not all the time. We monitored some of the critical functions in *SPEC CPU 2006* benchmark suite [9] and captured the values taken by their parameters. The result obtained is interesting and is shown in Table I. Some of the parameters are taking the same value across multiple function calls. Moreover, the idea of memoization presented in [18] is entirely based on functions with repeating arguments and they have listed more such functions. These all are indicating the possibilities of applying dynamic specialization.

Benchmark	Function	No. of calls [1]	Time (%)[1]	Unique Values
sphinx3	vector_gautbl_eval_logs3	2.3 M	26.83	1
hmmer	FChoose	277.2 M	1.88	1
sphinx3	subvq_mgau_shortlist	492.9 M	8.17	1
mcf	primal_bea_mpp	2.2 G	40.29	2

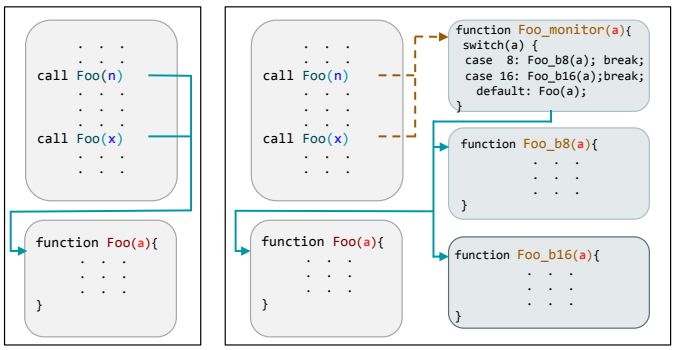
[1] `gprof` data

TABLE I: Repeatability of arguments

B. Our Approach

A function may be called from different parts of the program. Since the values taken by the parameters may differ in each call even from the same call site, it is not possible to directly replace the original function call by a call to a specific specialized version. Instead, different versions need to be maintained according to the arguments. In our approach, we use an extra function, called *monitor function*, to manage these specialized versions and redirect function calls to appropriate versions. The *monitor function*s, one for each function, are created dynamically and we replace all the original function calls by a call to the *monitor function*. Fig. 1 shows the difference in function execution before and after applying specialization. In normal execution of the program, the original function, `Foo`, is directly executed. But in the case of dynamic specialization the *monitor function*, `Foo_monitor`, is executed first and then the appropriate version is called from it.

Since the *monitor function* is not part of the original application, it creates some execution overhead on the application during each call to the original function. Dynamic function specialization can be beneficial only when the optimized versions gain enough speedup to overcome the overhead created by monitor function. Hence, before creating the specialized



a. Normal execution

b. Dynamic function specialization
(*All calls to function* `Foo` *are redirected to* `Foo_monitor` *and* `Foo_monitor` *decides which version to be executed*)

Fig. 1: Call sequence: Normal vs Specialization

versions, repeatability of the arguments must be ensured so that the optimized versions are expected to be executed more number of times compared to the original function. The values taken by the parameters of a function are observed initially for ensuring repetition before creating specialized versions. Specialized versions are created for such repeating arguments in parallel to the execution of the program.

The overall procedure is carried out in two phases, *Analyzing and Monitoring phase* and *Specialization phase*. Analyzing and monitoring phase is the decision-making phase in which the functions which are needed to be specialized are determined. In specialization phase, the different versions are created and included into the process. We now explain these two phases in detail.

Listing 2

```
double Sum(int a[], int b)
    for i = 1 to  b
        c += a[i];
return c;
```

1) Analyzing and Monitoring phase: The first step in this phase is to find out the suitable 'hot' functions for specialization. The hot functions in the application can be determined by monitoring the spots where the application spends most of its CPU time. Many techniques have been proposed, such detection can be performed with very low overhead [16]. Not all hot functions may be suitable for specialization. Suitable functions for specialization are chosen based on how the parameters are used inside the function body. If knowing the value of the parameter creates new possibility for applying different optimization techniques, like those discussed in section II, then the function can be marked as suitable. For example, consider the function `Sum` given in Listing 2. In this case the trip count of the loop can be precomputed once the value of `b` is known. And by knowing the trip count different optimization techniques can be applied efficiently [3], [6]. More details on choosing suitable function are given in Section IV.

Argument Value	Repetition Count	Target Function
8	3128	Foo
13	129	Foo
16	2451	Foo

TABLE II: Monitoring Table

The next step is to collect the actual values taken by the optimizable parameters of the functions. After a function is found suitable for specialization, a *monitor function* is created for it and all calls to it are redirected to this *monitor function*. The *monitor function* is used to collect arguments. It contains a table, as shown in Table II, which can store arguments, their repetition count and target function indicator. On every call to the function, depending on the argument, repetition count is incremented and corresponding target function is called. Initially all the entries of the *Target Function* column are set to the original function. The need of applying specialization is decided by the repetition count of the argument. If none of the arguments is repeating for a given amount of time, monitoring is disabled by restoring the original function calls. It can be re-enabled at a later time to capture a change in application phases.

2) Specialization phase: The specialized versions of the function are created based on the repetition count of the arguments. When the count of any of the arguments reaches a threshold value, a specialized version is created. We currently rely on the availability in the program executable file of an intermediate representation of the program generated during the compilation. The technique is sometimes referred to as *fat binaries* (see for example the recent work of Nuzman et al. [15]). The specialized version is produced by recompiling the intermediate representation after replacing the corresponding parameters with their values. The compiler can then apply constant propagation followed by all available optimization techniques, including hardware-specific optimizations, according to the parameter value.

The optimized versions are included in the process by injecting their binary code into the process memory space. The *Target Function* value in the corresponding table entry is then modified so that future calls to the function with this argument will execute the specialized version. For example, if we consider 1000 as the threshold value, then there is a possibility of making two specialized versions of `Foo` based on Table II entries. The specialized versions `Foo_b8` and `Foo_b16` can be created for the values `8` and `16` respectively and then the table entries are modified as in Table III.

Argument Value	Repetition Count	Target Function
8	3128	Foo_b8
13	129	Foo
16	2451	Foo_b16

TABLE III: Modified Monitoring Table

Special case: In the case of pure functions, such as the transcendental functions like `sin`, `cos`, `log` etc, by knowing the value of the parameters, we can directly calculate and store the result of the function as these functions are known to

return the same value across calls for the same argument(s). In such cases, it is not required to create a specialized version and instead we just need to store the result. This type of specialization is known as *function memoization* [18]. We consider such functions separately and create a special kind of *monitor function* with a separate table structure. In this table, we store only arguments and results as shown in Table IV. For each argument, *monitor function* executes the original function on first call and stores the result in the table and returns this result for subsequent calls with the same argument.

Argument Value	Result
5	32
8	256
16	65536

TABLE IV: Monitoring table for the pure function `exp2`

IV. Implementation Details

A. Overview

Our specialization approach includes four major tasks.

***Profile*:** Find out 'hot' functions of the application.

***Analyze*:** Choose 'hot' functions which are suitable for specialization.

***Monitor*:** Collect values taken by the parameters of suitable functions.

***Specialize*:** Create specialized versions of the function for repeating arguments and include them into the application.

The first three tasks are part of *analyzing and monitoring* phase and the last one is of *specialization* phase.

These tasks are performed by another program, named *Optimizer process*, which runs in parallel with the target application process. The target program remains unmodified, and does not even need to be restarted. The optimizer operates in a manner similar to a debugger, attaching to and detaching from its target. More details are given in Section IV-B.

To achieve our goal of dynamic function specialization, we need to make some changes to the application's memory space, like changing function call instructions and including new binary codes of the functions. For these, we use PADRONE, a library which provides APIs for dynamic binary analysis and optimization [16]. PADRONE has a monitoring functionality, in which for a function in the application, a *monitor function* can be injected to the application such that the *monitor function* executes first on every call to the original function. For executing the *monitor function* first, PADRONE catches all calls to the original function and redirects them to the *monitor function*. It catches the original function calls with the help of `trap` instruction by replacing the first instruction of the function with the `trap` instruction. Now on every calls to the function, the application gets stopped in the `trap`. PADRONE can then calculate the address of the call site by fetching the return address of the function from the stack and can change the call instruction at the call site accordingly. Note that the original function can also be called from the *monitor function*.

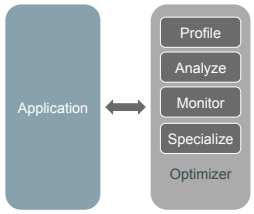


Fig. 2: Application and Optimizer Processes

In order to avoid trapping the calls from *monitor function*, PADRONE makes a copy of the original function and the calls from *monitor function* are redirected to the copy. Newly generated code is store in a code cache[1] in the application's memory space.

We use PADRONE also for finding the hot functions and for injecting specialized versions. Since it needs binary code of new functions as shared libraries for injecting to the running process, we compile our monitor functions and specialized versions into shared libraries. The hot functions of an application can be found out by using PADRONE's profiling functionality.

Since the available information in machine-level code is very limited, optimizing a binary code alone is very difficult. So we use, similar to [15], both LLVM intermediate representation [10] (LLVM IR) and binary code of the program. LLVM IR is used for creating optimized versions of the functions and the binary code is for executing the program. The LLVM IR is produced from the source code during the compilation of the program.

Although our current approach relies on fat binaries, we plan to drop this requirement by lifting binary code to LLVM IR. Previous work by Hallou et al. [7] using the McSema infrastructure [5] showed that this is a viable path, including for optimizations as complex as vectorization.

B. Optimizer Process

We implement the idea of dynamic function specialization with the help of our *Optimizer process* which runs alongside the application as shown in Fig. 2. The specialization process is carried out in this separate process (thanks to PADRONE) to reduce the impact on the target application (at least on a multicore, or a processor equippe with simultaneous multithreading such as Intel's HyperThreading).

The optimized versions of a function are created from the LLVM IR of the program produced during the compilation of the source code. We write LLVM *passes* [1], [10] for creating optimized versions and *monitor function* of a function and also for finding the suitable functions for specialization among the critical ones. We use three different passes.

***isPossible pass*:** Used in *analyzing* stage to check the suitability of a function for specialization.

[1]A code cache is a contiguous memory area in the application's memory heap. PADRONE creates it by writing a small piece of code into the application's code segment which calls `malloc` to allocate the required memory.

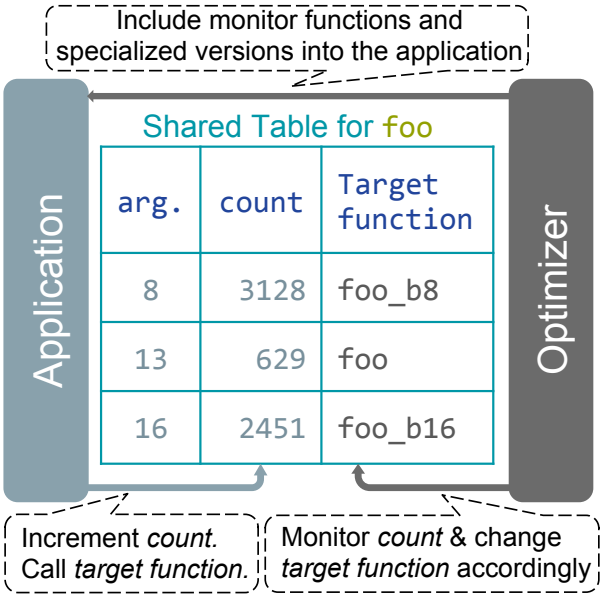


Fig. 3: Both application and optimizer processes access the shared table

***monitor pass*:** Used in *monitoring* stage to create *monitor function*

***optimize pass*:** Used in *specialization* stage to create specialized versions of a function.

The *optimizer process* starts its execution just after the application process starts running. Then, PADRONE is used to attach it to the application process. The detailed explanation of each stage of its execution is given below.

Profiling Stage: Instead of specializing every function in an application blindly, we apply specialization only on critical functions in it. We use profiling, a feature provided by PADRONE, to find out these critical functions. PADRONE probes the performance counters of the processor at regular interval of time with the help of *Linux perf event* subsystem kernel interface. Each probe provides a sample and we use these samples to figure out the critical functions by inspecting the instruction pointers included in each sample. One session of profiling lasts only a few seconds and it repeats at regular intervals of time to catch more live critical functions.

Analyzing Stage: All functions may not be suitable for specialization. Functions with no parameters may only benefit from a hardware-specific optimization, but not from specialization. Since our *monitor function* creates some overhead, the specializable parameter of the function need to be used in a critical part of the function code such that specialization should not result in a slow down to the overall process. Our cost-model attempts to capture all these phenomena. Since, loop optimizations heavily impact the performance of a program, a parameter which is part of the loops can be a good candidate for specialization. Parameters of pure functions are also considered since the entire function call can be reduced to a constant.

The *isPossible* pass is used to find out functions which are suitable for specialization. In this pass, we analyze each uses of integer or floating point type parameters of the function. Once we find a use in an expression calculating trip count of a loop, we backtrace the uses of other operands in the expression. If the other operands of the expression are derived either from integer or floating point type arguments or from constants, then we mark the function as specializable. Currently we are looking only for arguments of `int`, `long int`, `float` and `double` data types but this can be extended to other data types. This pass outputs the name and data types of the parameters which satisfy our condition and a flag indicating whether the function returns `void` or not. The list of all functions which are suitable for specialization could also be made earlier, during creation of the LLVM IR of the program, and can then be used at run time to pick out specializable critical functions. This would minimize the overhead, but we have not explored that direction yet.

Monitoring Stage: For each suitable function, we create a *monitor function* for collecting arguments and redirecting function calls to appropriate versions. The arguments are stored in a look up table inside *monitor function*, as in Table II, to detect the possibility of specialization. The *monitor function* performs two operations.

1) Increment *repetition count* corresponding to the argument
2) Call the corresponding *Target function* and return its result.

The table is hosted in a System V shared memory segment, and shared between the two processes as shown in Fig. 3. The application process, through *monitor function*, updates the first two columns of the table while the third column is modified by the *optimizer* process. Initially the *Target Function* column of all the table entries are set to original function and they are modified accordingly on creation of each specialized version. A hash function is used to index the table. In order to perform a quick table look up, we use the folding `XOR` based indexing as used in [18]. The idea is, higher order and lower order bits of the argument are repeatedly `XOR`ed until we get a 16 bit number. And we mask 4 bits of it to get a 12 bit number which can be used as index to a table of size 2^{12}. In case of a conflict, we directly call the original function without modifying the table entry. For functions with more than one (specializable) argument, first we `XOR` all arguments to a single one and then applies the above procedure on it.

After the LLVM IR of *monitor* function is created, it is compiled to a shared library which can be dynamically linked. Then, with the help of PADRONE this shared library is injected into the process and subsequent calls to the original function are trapped and redirected to the monitor function. Now on, all the values taken by the specializable parameters are collected in the table inside *monitor* function which is used in *specialization* stage.

Specialization Stage: The need of specialization is decided by looking the values taken by the parameters of the function so far. The values and their repetition counts are stored in shared tables. These tables are analyzed by optimizer process at regular intervals. A specialized version of a function is

created for a particular argument, when the repetition count of that argument crosses a threshold value. We set the threshold value for a function to 10% of its total number of calls so far or to 500, whichever is greater.

The *optimize* pass is used to create the LLVM IR of the specialized version from the LLVM IR of the program. This pass makes a copy of the original function with the name of specialized version. Then all the uses of specializable arguments are replaced with their exact value. The resulting LLVM IR is then compiled to get the executable file of the specialized version. The compilers apply all the suitable optimization techniques based on the argument value. It is also possible to create more hardware specific versions from the LLVM IR [8]. The specialized version is then injected to the process using PADRONE and the value of *Target Function* in the corresponding table entry is changed to the specialized one as shown in Table III.

If none of the arguments are found repeating more than the threshold value even after the function is executed more than 50k times, then the function is removed from the suitable functions list and it will not be considered for monitoring anymore.

We repeat these four stages until the application finishes its execution. In some cases where the function has more than one suitable argument, it is also possible to have only a subset of them to be repeating. In such cases, we may need to create the *monitor* function once more by considering only the repeating arguments.

Additional benefit: As mentioned earlier, a compiler can apply more optimizations to a program, once it knows the hardware details of the running machine. Our specialized versions are optimized based on the hardware. However, before creating specialized versions, we execute the original function. So it is beneficial to produce also a hardware-specific version of the original, non-specialized, function.

Handling pure functions: The library functions, like `cos, sin, exp` etc, are considered separately. In such cases, we make only *monitor function* and it can store the result in the table. So, we do not need to create specialized versions. To redirect the original function calls, PADRONE updates the GOT table entry with the address of *monitor function*s. We need to execute the original function for the first call of each argument value to get the result. Since the GOT table entry is modified, we cannot directly call the function by its name from the *monitor function*. Therefore we use `dlsym` to fetch the address of the function and we call the function by its address.

V. Example

This section illustrates how our optimizer process may impact the execution time of a function with the help of a simple example. We used the function given in Listing 3. We call it from an infinite loop, and we measured its execution time on each call. Fig. 4 reports our observation. The x-axis represents the time elapsed and the y-axis represents the execution time (average of five consecutive calls) of the function. Before specialization, it takes around 2.13 seconds to complete the execution with the values 9999 and 2018 to the arguments p and q respectively. At the 180^{th} second, we started a fairly aggressive profiling period for 100 seconds with a frequency of 1000 samples per second. This is visible as a small bump on the graph. During this profiling session, the optimizer process identifies *foo* function as a hot function and starts monitoring its parameters. The monitoring continued until the repetition count reaches the threshold. At around the 730^{th} second the count crosses the threshold and a specialized version of *foo* is created. The execution time drops to around 2.04 seconds only to complete its execution. Fig. 4 shows a slight increase in the execution time during profiling and monitoring stages, but on the long run, specialization is clearly able to recoup this overhead.

Listing 3 A specializable function

```
long int foo(int a[], int b[], int p, int q)
    long int i, j, k=0;
    for j = 0 to (p+q)*2
        for i= 1 to q*2
            k+= ((a[i]/b[i])) % 2000;
            k = k/(p*q-9998*q-2047);
return k;
```

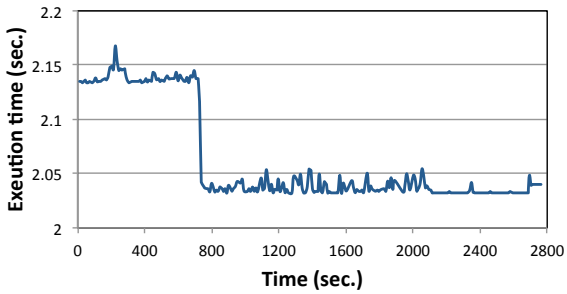


Fig. 4: Impact of specialization on execution time of a function (lower is better)

VI. Result

A. Experimental set-up

We implemented our idea of dynamic function specialization on an Intel Broadwell core i7 architecture with $4\,MB$ L3 cache and $16\,GB$ RAM. We set the clock speed to $2.7\,GHz$. The LLVM version used is 3.7.0 with $O3$ optimization on a Linux version 3.19 operating system.

Benchmarks: Since PADRONE is tested only for `C` language programs, currently we implemented dynamic function specialization only for programs written in `C` language. We choose benchmarks, written in `C`, which contain functions satisfying our specialization criteria which are discussed in Section IV-B. The first one is the function should be critical and it should contains at least one integer or floating point type argument. The second one is this argument should be used in trip count calculation and the argument should repeat. We

report only on qualifying benchmarks, since others practically show neither speedup nor slowdown: *hmmer* and *sphinx3* benchmarks from SPEC CPU 2006 benchmark suite, *equake* benchmark from SPEC OMP 2001 benchmark suite [2] and *ATMI* [13].

Prerequisites: Our optimizer process runs in parallel with the application process. We need both binary executable and LLVM IR of the application. Since all our decisions regarding the specialization are taken at run time, we do not require any prior information about the application.

B. Overhead

Among the four stages of our implementation, profiling, monitoring and specialization stages directly affect the execution of the application program. For analyzing the slowdown caused by our implementation, we run it in two different situations. In first, both monitoring and specialization stages are disabled and in second, only specialization is disabled. We use *h264ref*, *hmmer*, *sphinx3* and *gobmk* benchmarks from SPEC CPU 2006 for analyzing the overhead. The result is shown in Fig. 5. All other benchmarks may also perform similarly if there is no specializable functions found. The profiling stage is repeated 3 times. The profiling sessions are carried out for 5 seconds with 100 samples per second, for 10 seconds with 200 samples/second and for 20 seconds with 400 samples/second respectively. And the overhead created by profiling is very less (less than one second). The overhead created by monitor stage depends on the number of times the functions are called. On each function call, we have a table look up and an extra function call to monitor function. So the overhead created by monitor stage is different for different benchmarks. Considering the two extremes in Fig. 5,

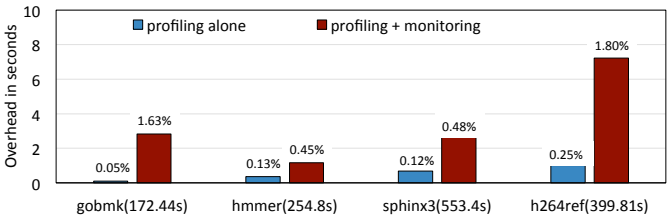


Fig. 5: Overhead by profiling and monitoring

Normal execution time of the benchmark (in seconds) is given in brackets and the percentage of overhead is given on top of each bar.

the average number of calls of monitor function in h264ref is around 116 million while for hmmer, it is only 458 thousand. The overhead can be reduced by calling a machine dependent optimized version of the original function instead of the actual one in the case of non repeating arguments. For analyzing the overhead, we used the actual version included in the binary of the program itself.

C. Speedups

Fig. 6 reports the speedups we obtained. We are including only the benchmarks in which we can specialize at least one function. All other benchmarks may result similarly in Fig. 5. The functions `subvq_mgau_shortlist` from *sphinx3* and `primal_bea_mpp` from *mcf* given in Table I are not specialized because the former one contains a `static` variable and in the later one the repeating argument is not part of the loop. Currently our implementation is not handling functions with static variables.

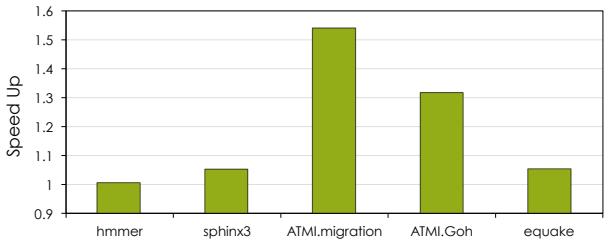


Fig. 6: Speedups

The total time taken for the application is measured using `time` command and the speedup shown in Fig. 6 are measured based on the total time taken by the application. In *hmmer* benchmark, functions *P7Viterbi* and *FChoose* are monitored. But the monitored argument in *P7Viterbi* function is not repeating. In FChoose function, one integer argument is repeating and it takes 20 as its value all the time. The compiler applies *loop unrolling* technique on a loop inside this function to get the optimized version. Since FChoose is taking only 2 % (see Table I) of total execution time of the application, the improvement in this benchmark is minimum. In sphinx3 benchmark, function `vector_gautbl_eval_logs3` is monitored and specialized and we obtain around 5 % improvement at runtime. In this function, two integer arguments are repeating with the values 0 and 4096 on every call to the function. Here, compiler applies *vectorization* technique to improve the run time of the specialized version.

We also obtain good improvements in ATMI application: 35 % and 24 % for examples ATMI.migration and ATMI.Goh respectively. In both cases, we specialized the Bessel functions $j0$ and $j1$. And in equake, we obtain an improvement of 5 % thanks to the sin and cos functions.

VII. Related Work

This section discuss about a number of works in dynamic function optimization.

a) JIT technology with C/C++: Feedback-directed dynamic recompilation for statically compiled languages: [15] discusses about dynamic optimization by using both native executable file and intermediate representation (IR) file of the program. During execution, they recompile hot methods from IR file using a Java JIT compiler and the recompiled versions are stored in a *Code Cache*. With the help of a trampoline created at the beginning of the original function body, the function calls are redirected to the new recompiled version. This work is not about applying function specialization, but we both use almost similar recompilation technique to create different optimized versions. And this work shows that, even without knowing the value of a parameter, a more optimized version of a function can be produced dynamically.

b) Intercepting Functions for Memoization: A Case Study Using Transcendental Functions: [18] discusses about implementing *memoization*, saving the result of execution of a section of program for future use, in software for *pure* functions. The idea is saving the result of a function in a table according to its arguments and return these results, instead of executing the function again, in future calls. The paper shows that a good amount of application have argument repetition in functions even after applying state of the art compiler optimization. But that paper is not studying if any particular value or class of values are being repeated and if we could do optimization based on them. Our work is an extension of this work. We used both memoization in case of mathematical functions and specialization for other functions. Memoization works when the function can be entirely reduced to a constant but specialization can consider intermediate steps: not constant, but some computations are eliminated/simplified.

c) Just-in-Time Value Specialization: [17] discusses about creating more optimized version of a function, based on runtime argument value, for JavaScript programs. JavaScript programs are usually distributed in source code format and compiled at the client-side. A just-in-time compiler is used to compile JavaScript function just before it is invoked or while it is being interpreted. In just-in-Time value specialization, they observed that most of the JavaScript functions are called with the same argument set. So they replaced the parameters of the function with the values while compiling at client side to get native code. But, if the function is called with a different argument, the specialized native code is discarded and actual source code is compiled. Then the function is marked so that it won't be considered for the specialization in future. The main difference with our work is that, they are not handling multiple versions of a function. Instead, at a time there is only one version of a function, either the specialized version or the original one. And that version is created directly from the source code.

d) Tempo: [4] Tempo is a specializer tool developer specific to the C language. It provides a declarative language for the developer using which he can provide specialization options for the tool. it has both compile time as well as run-time specialization options. The major difference of this work and ours is that we do not require any help from the programmer and hence our technique can be applied to any existing program. Though due to implementation limits we could only use C programs for our results our technique is more general and is extendable to any other language.

VIII. Conclusion

Compilers can do better optimization with the knowledge of run-time behaviour of the program. We propose a runtime optimization technique called *dynamic function specialization*. We analyze the values of arguments of a function and create different versions for all candidate functions in parallel with the execution of the program. Then we inject these specialized versions into the running process with the help of PADRONE library. The function calls are redirected to the appropriate versions with the help of an extra function. Our approach does not require restarting the application. Our speedups range from 1 % to 35 % for a mix of SPEC and scientific applications.

Our current implementation relies on *fat binaries* which store the compiler intermediate representation of function in the program executable. Future work will consist in lifting binary code to LLVM IR, opening the door to optimization of any program, including legacy or closed-source.

References

[1] Writing an LLVM pass. http://llvm.org/docs/WritingAnLLVMPass.html. Accessed: 2016-09-26.

[2] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. *SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance*. Springer Berlin Heidelberg, 2001.

[3] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994.

[4] Charles Consel, Julia L. Lawall, and Anne-Franoise Le Meur. A tour of Tempo: a program specializer for the C language. *Science of Computer Programming*, 52(13), 2004. Special Issue on Program Transformation.

[5] Artem Dinaburg and Andrew Ruef. McSema: Static translation of x86 instructions to LLVM. https://blog.trailofbits.com/2014/06/23/a-preview-of-mcsema/. Accessed: 2016-11-02.

[6] J. J. Dongarra and A. R. Hinds. Unrolling loops in fortran. *Software: Practice and Experience*, 9(3):219–226, 1979.

[7] Nabil Hallou, Erven Rohou, and Philippe Clauss. Runtime vectorization transformations of binary code. *International Journal of Parallel Programming*, pages 1–30, 2016.

[8] Nabil Hallou, Erven Rohou, Philippe Clauss, and Alain Ketterlin. Dynamic Re-Vectorization of Binary Code. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation - SAMOS XV*, Agios Konstantinos, Greece, July 2015.

[9] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[10] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO'04)*, 2004.

[11] Jason Mars and Robert Hundt. Scenario based optimization: A framework for statically enabling online optimizations. In *International Symposium on Code Generation and Optimization (CGO)*, 2009.

[12] Hidehiko Masuhara and Akinori Yonezawa. A portable-approach to dynamic optimization in run-time specialization. *New Gen. Comput.*, 20(1):101–124, January 2002.

[13] Pierre Michaud, André Seznec, Damien Fetis, Yiannakis Sazeides, and Theofanis Constantinou. A study of thread migration in temperature-constrained multicores. 4(2), June 2007.

[14] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[15] Dorit Nuzman, Revital Eres, Sergei Dyshel, Marcel Zalmanovici, and Jose Castanos. JIT technology with C/C++: Feedback-directed dynamic recompilation for statically compiled languages. *ACM Trans. Archit. Code Optim.*, 10(4):59:1–59:25, December 2013.

[16] Emmanuel Riou, Erven Rohou, Philippe Clauss, Nabil Hallou, and Alain Ketterlin. PADRONE: a Platform for Online Profiling, Analysis, and Optimization. In *DCE 2014 - International workshop on Dynamic Compilation Everywhere*, Vienne, Austria, January 2014.

[17] Henrique Nazare Santos, Pericles Alves, Igor Costa, and Fernando Magno Quintao Pereira. Just-in-time value specialization. In *International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, USA, 2013. IEEE Computer Society.

[18] Arjun Suresh, Bharath Narasimha Swamy, Erven Rohou, and André Seznec. Intercepting Functions for Memoization: A Case Study Using Transcendental Functions. *ACM Transactions on Architecture and Code Optimization (TACO)* , 12(2):23, July 2015.

[19] Scott Thibault, Charles Consel, Julia L. Lawall, Renaud Marlet, and Gilles Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3), 2000.

[20] John Whaley. Dynamic optimization through the use of automatic runtime specialization, 1999.